

Interface Building for Software by Modular Three-Valued Abstraction Refinement

Pritam Roy
pritam@ee.ucla.edu

Computer Science and Electrical Engineering Department
University of California, Los Angeles, USA

Abstract. Verification of software systems is a very hard problem due to the large size of program state-space. The traditional techniques (like model checking) do not scale; since they include the whole state-space by inlining the library function codes. Current research avoids these problem by creating a lightweight representation of the library in form of an *interface graph* (call sequence graph). In this paper we introduce a new algorithm to compute a safe, permissive interface graph for C-type functions. In this modular analysis, each function transition is summarized following three-valued abstraction semantics. There are two kinds of abstraction used here. The global abstraction contains predicates over global variables only; however the local abstraction inside each function may also contain the local variables. The abstract summary needs refinement to guarantee safety and permissiveness. We have implemented the algorithms in TICC tool and compared this algorithm with some related interface generation algorithms. We also discuss the application of interface as an offline test-suite. We create an interface from the model program (specification) and the interface will act as a test-suite for the new implementation-under-test (IUT).

1 Introduction

Verification of software systems is a very hard problem due to the large size of program state-space. Most software programs contain library functions and these kind of functions are examples of *open systems*. The verification of such open systems becomes infeasible due to two main problems. Firstly, in order to verify a given program one needs to *inline* the library function code and it increases the space complexity of the verification algorithms. Current formal techniques like model-checking can not handle the large state-space generated from the program variables. The second option is to verify the library functions a priori so that there is no need to inline them. For this purpose, most of the time a small code containing a sequence of library functions calls (called *client*) is written. The client code invokes the library functions to close the open system. The library functions are impossible to verify in the absence of exhaustive client program. Hence most of the verification approaches plug-in a client code to close the open-system.

1.1 Interface and Properties

The current research [9,1,3] avoids these two problems by applying *modular verification* techniques which builds a small call sequence graph, called *interface* representing union of all client programs. The interface contains all possible call sequences which leads the library to error or illegal states. Similarly, the interface should contain all possible call sequences which avoids the error states. Henceforth constrains on the use of the library function calls from outside and the user can distinguish the legal call sequences from the illegal ones by simply looking at the interface. There are two immediate benefits of using the interfaces. Firstly, these interfaces are light-weight representation of the libraries and the implementation of the library functions can be replaced by the interface. Secondly, the interfaces can be constructed without the help of any client program. The interface should be *safe* i.e. all illegal call sequences (which leads the library to the error states) will be present in the interface. The interface graph should be *permissive* i.e. all legal sequences will be present in the interface.

1.2 Related Work

However, there are some challenges in building succinct interfaces. The interface size can become exponential in terms of number of variables. A symbolic representation and abstraction techniques partition the state-space into a small number of regions where every region represents one node of the interface graph. Some researches apply these abstraction and symbolic techniques to obtain a small but safe and permissive interface.

The work by Alur et. al. ([1]) uses *Angluin's learning algorithm* L^* to create an interface. The algorithm learns the interface language by asking membership and equivalence queries to teacher (here program). The generated interface is safe and minimal; but not permissive. To handle big case studies predicate abstraction has been used, however the user need to provide the predicates. There is no automatic abstraction refinement. The algorithm returns minimal size interface if the algorithm is not hit by timeout. Experimental results show that even in small examples timeout occurs. The CEGAR approach by Henzinger et. al. ([9]) creates a safe and permissive interface. The size of the interface can be big enough depending on the chosen counter-example. The direct approach by Beyer et. al. ([3]) creates an interface which is safe and permissive. This approach does not use abstraction and hence the interface can become very large.

1.3 Contribution

Unlike the related work, our work can also be used in unstructured or non-object oriented (C style) functions. In an object-oriented framework every class variable is accessible to every class method and can be a global variable to the class method. Instead we assume that each function may contain several local variables in addition to those global variables. Hence, we have *more general platform* to compute interface. Each of these functions can also have several sequential

updates of variables, call to other functions even recursive calls to themselves. However, we compute the interface including only functions accessible to the user level.

In the first stage of three stage algorithm, every C library function is parsed by CIL (C Intermediate Language)[11] and converted into TICC [4] input language. This language syntax is similar to the guarded-update language. We have implemented the next two stages in this Multi-valued Decision Diagram [10]-based symbolic tool TICC. The second stage computes the transition summary of each function. This modular algorithm handles each function separately including local variables within the scope. However, the space complexity of function summary becomes a bottleneck in order to compute big functions which may contain large number of guarded-updates. Hence, we employ three valued abstraction refinement schemes in addition to symbolic techniques. The abstraction in summarization ensures small size; whereas successive refinement of the abstract states fine tune the abstraction to obtain the safety and permissiveness. In the last stage, an interface graph is built from the abstract set of states. We show different stages of building a symbolic safe and permissive interface in the following example.

Example 1 (Motivating Example). Figure 1(a) defines a stack data-type *stackT* and two functions *push* and *pop*. The data type *stackT* has an array of integers *el* of size *MAX* and an integer showing the *top* of the stack. The function *pop* returns error when the stack is empty i.e. *top* is zero. The function *push* returns error if the *top* is equal to *MAX*. Otherwise copies the input value *sd* into the *el* array at address *top*. The *top* is incremented later. Figure 1(b) shows how the C code is converted into guarded-update rule in the next stage. The global variable *err* denotes the error in the library and the library goes to error state when *err* is set to 1. Figure 1(c) shows the interface graph from the set of rules. The initial state of the interface graph is *state 1* where the stack is empty. A call to *pop* function from the initial state will move the library into an *ERROR* state. Similarly calling *push* from state 3 will be an error due to full stack. We can note that the interface can create many legal as well as illegal sequences of stack functions. To check each of them we otherwise need a set of client programs.

Finally we discuss the applications of the safe and permissive interface graph. Firstly, any given client program can immediately verify with the help of the interface graph whether the function call sequence in the client leads the library to some error states. Secondly, the interface can actually provide an offline test-suite for a set of functions. Often the source of the library is unknown; however one can create a model program from the available documentation of the functions. The interface graph obtained from the model program can be used to test the implementation-under-test (IUT).

2 Preliminary Definitions

In this section we provide preliminary definitions and the background work.

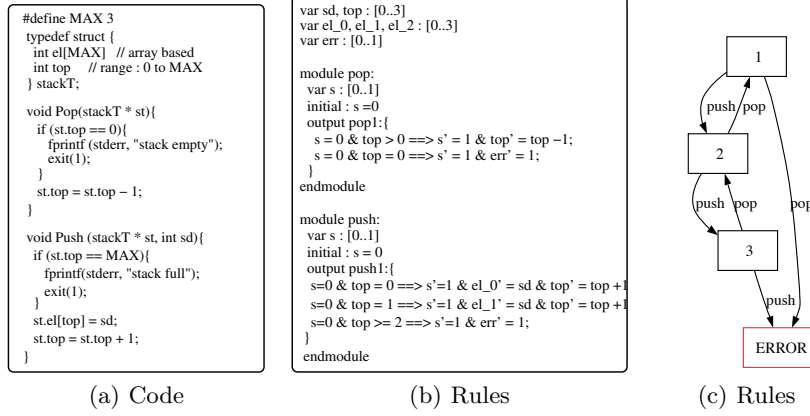


Fig. 1. Stack Example

2.1 A Transition System Model for Libraries

A software library module $Lib = (F_G, V_G, E, I)$ contains a set of functions F_G and a set of global variables V_G . The global variables V_G constitute variables declared outside any of the functions in F_G . The *global state space* S_G can be defined with respect to different valuations of global variables V_G . The variable $err \in V_G$ is a special global variable in Lib which can take two values 0 and 1. The library reaches an error set $E \subseteq S_G$ when the global variable err is set to 1. Moreover, the error set is a sink set of the library. The initial configuration of the library is given by set $I \subseteq S_G$.

Each function $f \in F_G$ also contains a set of local variables V_L^f . The *scope* of any local variable $v \in V_L^f$ is function f . There is a special local variable, called s , in V_L^f which corresponds to the relative location in the function with respect to the first location. For a function f , all variables V^f can be given as $V_L^f \cup V_G$ and *function state-space* S_f can be defined with respect to different valuations V^f . We note that each global set $s_G \in S_G$ is a non-empty subset of $s_G \subseteq S_f$ function state-space. The *initial local state set* $I_L^f \subseteq S_f$ denotes the entry point to the function f . All variables of the library Lib is denoted by V and is given by $V := V_G \cup \bigcup_{f \in F_G} V_L^f$. The *total state-space* S can be defined with respect to different valuations of all variables V .

Each function $f \in F$ contains some number (say k) of guarded-update rules. For i -th such rule, its condition part $i.guard \subseteq S_f$ can be given as a set of function states, and the assignment part $i.update \subseteq S_f \times S_f$ can be given as the set of transitions. For a set $X \subseteq S_f$, $i.update(X) : S_f$ denotes the next state of X in the i -th update rule. The conditional transition of rule i given as

$$i.trans := \{(s_1, s_2) \in S_f \times S_f \mid s_1 \in i.guard, s_2 \in i.update(i.guard)\}.$$

The transition relation $Trans^f \subseteq S_f \times S_f$ can be given as the union of rules corresponding to the function f i.e. $Trans^f := \cup_{i=1\dots k} i.trans$. We will use $Trans^f(t) \subseteq S_f$ to denote the successor set of state $t \in S_f$.

For a binary relation $\bowtie \in \{=, \leq, \geq\}$ and a state-space S , the set $S \mid_{v \bowtie a}$ denotes the set where the value of a variable v related to value a with relation \bowtie . For a set $X \subseteq S_f$, we define $support(X) \subseteq V_f$ as the set of variables whose value change result in a value change of X . Formally we can write,

$$support(X) := V^f \setminus \{v \in V^f \mid \forall s, s' \in S_f. s =_v s' \rightarrow s \in X \iff s' \in X\}$$

where $s =_v s'$ implies that $s = s'$ except for a variable $v \in V^f$. Interface graph is an input-enabled interface automata. Given a Library $Lib = (F_G, V_G, E, I)$ and global state-space S_G , we can define *interface-graph* or call sequence graph as $IG = \langle N, T, T_e, In, Er \rangle$ where,

- the nodes $N \subseteq 2^{2^{S_G}}$ correspond to the set of states,
- the set $In \subseteq N$ denotes the initial nodes corresponding to I ,
- the set $Er \subseteq N$ denotes the error nodes corresponding to E ,
- the set $T \subseteq N \times F_G \times (N \setminus Er)$ denotes good transitions.
- the set $T_e \subseteq N \times F_G \times Er$ denotes erroneous transitions.

2.2 Three Valued Abstraction

For a library $L = (F_G, V_G)$, a function $f \in F_G$ and a function state-space S_f , an *abstraction* $R \subseteq 2^{S_f \setminus \emptyset}$ is defined such that each abstract state (or region) $r \in R$ is a non-empty subset $r \subseteq S_f$ of concrete states. We require $\bigcup R = S_f$. For subsets $T \subseteq S_f$ and $U \subseteq R$, we write:

$$U \downarrow = \bigcup_{u \in U} u \quad T \uparrow_R^m = \{r \in R \mid r \cap T \neq \emptyset\} \quad T \uparrow_R^M = \{r \in R \mid r \subseteq T\}$$

Thus, for a set $U \subseteq R$ of abstract states, $U \downarrow$ is the corresponding set of concrete states. For a set $T \subseteq S_f$ of concrete states, $T \uparrow_R^m$ and $T \uparrow_R^M$ are the set of abstract states that constitute over and under-approximations of the concrete set T . We say that the abstraction R of a state-space S_f is *precise* for a set $T \subseteq S_f$ of states if $T \uparrow_R^m = T \uparrow_R^M$.

2.3 μ -Calculus

We will express our algorithms for solving reachability on the function state space in μ -calculus notation [8]. Consider a procedure $\gamma : 2^{V^f} \mapsto 2^{V^f}$, monotone when 2^{V^f} is considered as a lattice with the usual subset ordering. We denote by $\mu Z. \gamma(Z)$ (resp. $\nu Z. \gamma(Z)$) the *least* (resp. *greatest*) *fix-point* of γ , that is, the least (resp. greatest) set $Z \subseteq V$ such that $Z = \gamma(Z)$. As is well known, since V is finite, these fix-points can be computed via Picard iteration: $\mu Z. \gamma(Z) = \lim_{n \rightarrow \infty} \gamma^n(\emptyset)$ and $\nu Z. \gamma(Z) = \lim_{n \rightarrow \infty} \gamma^n(V)$.

2.4 Predecessor Operators

For a library function f and a function state-space S_f , we define the *one-step predecessor operator* $Pre^{f,1} : 2^{S_f} \mapsto 2^{S_f}$ as follows, for all $Y \subseteq S_f$:

$$Pre^{f,1}(Y) = \{x \in S_f \mid Trans^f(x) \cap Y \neq \emptyset\} \quad (1)$$

We define the *multi-step predecessor operator* $Pre^{f,*} : 2^{S_f} \mapsto 2^{S_f}$ as follows, for all $Y \subseteq S_f$:

$$Pre^{f,*}(Y) = \{s \in S_f \mid s \cap (\mu X.(Y \cup Pre^{f,1}(X))) \neq \emptyset\} \quad (2)$$

Intuitively, the set $Pre^{f,*}(X)$ consists a subset of S_f from which one can reach to X by applying zero or more transitions within the function f by applying rules one after another.

For the abstract state space R , we introduce abstract versions of $Pre^{f,R}$. As multiple concrete states may correspond to the same abstract state, we cannot compute, on the abstract state space, a precise analogous of $Pre^{f,R}$. We define two abstract operators: the *may* operator $Pre_m^{f,R} : 2^R \mapsto 2^R$, which constitutes an over-approximation of Pre^f , and the *must* operator $Pre_M^{f,R} : 2^R \mapsto 2^R$, which constitutes an under-approximation of Pre^f [6]. We let, for $U \subseteq R$:

$$Pre_m^{f,R}(U) = Pre^{f,*}(U \downarrow) \uparrow_R^m \quad Pre_M^{f,R}(U) = Pre^{f,*}(U \downarrow) \uparrow_R^M. \quad (3)$$

The fact that $Pre_m^{f,R}$ and $Pre_M^{f,R}$ are over and under-approximations of the predecessor operator is made precise by the following observation: for all $U \subseteq R$ we have

$$Pre_M^{f,R}(U) \downarrow \subseteq Pre^{f,*}(U \downarrow) \subseteq Pre_m^{f,R}(U) \downarrow \quad (4)$$

. For an integer $k \geq 1$ and function state-space S_f , we recursively define the *k-step post operator* $Post^{f,k} : 2^{S_f} \mapsto 2^{S_f}$ as follows, for all $X \subseteq S_f$:

$$Post^{f,1}(X) = \cup_{x \in X} Trans^f(x) \quad (5)$$

$$Post^{f,k}(X) = Trans^f(Post^{f-1,k}(X)) \quad (6)$$

For an abstract state space $R \subseteq 2^{S_f}$, we define the *abstract post operator* $Post_m^{f,R} : 2^R \mapsto 2^R$ as follows, for all $X \subseteq R$:

$$Post_m^{f,R}(X) = \{r \in R \mid r \cap Post^{f,k}(I_L^f \cap (X \downarrow)) \neq \emptyset\} \quad (7)$$

where k is the smallest integer to satisfy $Post^{f,k+1}(I_L^f \cap (X \downarrow)) = \emptyset$. Intuitively, the condition implies that no new states are added in the $k+1$ -th iteration, hence the last updated value when f returns can be obtained by applying $Post^{f,k}$ to a subset of $X \downarrow$ corresponding to the function's initial state set I_L^f .

3 Translation from C to Guard-Update Rules

In this section we discuss our procedure to convert C functions into the "sociable interface automata" [5] format. This format contains several guarded-update rules and is the input format of our symbolic tool TICC. In our work the front-end and back-end are separate. Hence one only need a different front-end to parse functions from any other language (like Java/C++) to generate the TICC input format models. The next stages of the algorithm can reuse the out tool TICC to build interface graphs.

The C functions are fed into CIL[11] tool which parses C source code and returns the control flow graph. The control flow graph contains block structure as nodes and the conditions as the transitions. We have modified the control flow graph for each function into set of guarded-update rules. The conditions are represented as guards and the assignments are represented as updates. The special local variable s defines the location of current block. For a variable v , the primed variable v' denotes the v in the next sequential step. When the translator encounters a critical error condition (e.g. call to *exit*(1)) in the control flow graph; the global variable *err* is set to 1 in the translated library.

- Control Flow Structures: The C source like "if (a =0) {b=0;} else {b=1;}" is converted into the following rules:

$$\begin{aligned} a = 0, s = 0 &\implies b' = 0, s' = 1; \\ a! = 0, s = 0 &\implies b' = 1, s' = 1 \end{aligned}$$

The switch and loop (like while, for) structures can be handled similarly.

- Variables and Data Structures: Currently the algorithm supports unsigned integers with small number (e.g. 4) of bits. The fixed-size arrays and structures are flattened in the translation process. In the Integer Stack example in Figure 1(b) shows how an array of size 3 is translated as 3 integer variables. The structure elements are also flattened in the example. Currently our translation does not directly handle pointers and recursive data types. However we can manually translate the pointers into integers only if we know that the control flow of the function does not depend on the value at its pointer location.
- Function Calls: Currently in order to compute the abstract transition for function f , we inline all the intermediate function call inside the body of f . In the guarded-update rule semantics, the rules of the intermediate functions are explicitly added to the rules of f . An explicit stack data structure is added to store the return address and the context variables. This trick can be applied to one function calling another function as well as the non-tail recursive function calls. The tail-recursive function calls can be converted into loops and do not need the stack. In the Appendix, we show a complete translation of a recursive c function.

4 Algorithm

In this section we assume that the C functions are already parsed by CIL and modified into a software library module $Lib = (F_G, V_G, E, I)$. We describe the basic algorithms for abstract refinement and building interface from a given library Lib . We also provide some implementation specific optimizations.

4.1 Basic Algorithm

Algorithm 1 computes the interface for library $Lib = (F_G, V_G, E, I)$. The algorithm takes as input the library Lib , a set of functions $F \subseteq F_G$, an abstraction R . The first abstraction is obtained from the error set E and initial set I . Let us define $r_1 = \{s \in S_G \mid s \in E\}$, $r_2 = \{s \in S_G \mid s \notin E, s \in I\}$ and $r_3 = \{s \in S_G \mid s \notin E, s \notin I\}$. For $i \in \{1, 2, 3\}$, if r_i is non-empty, then we add the set to R as one of the initial abstract states. The algorithm 1 calls $AbsRef$ for every function $f \in F$ separately to obtain a refined abstraction R w.r.t. the function. The procedure $BuildInterface$ returns an interface graph IG given the set of abstract states.

Algorithm 1 Explore(Lib, F, R)

Input: a library $Lib = (F_G, V_G, E, I)$, set of functions F , abstraction R

Output: Interface Graph IG

1. **for each** $f \in F$ **do** $R := AbsRef(R, f, E)$ **end for**
 5. $IG := BuildInterface(R, F, Lib)$
-

Modular Verification : Each function is considered separately in $AbsRef$ (Algorithm 2). Since, the interface graph is an input-enabled interface automata, every abstract state in the function can be checked separately for error reachability in one step function transition. The algorithm starts with the initial abstraction R and the set of useful variables V_{abs} are obtained from the support set of the abstract states. The local abstraction R_f and global abstraction R_G are initialized with R . The must abstraction transition is computed with respect to R_f and we compute the must predecessor S_M of the error set E . The set S_M determines the set of states of the function which eventually reach the error set E . The set S_M^f is subset of S_M corresponding to the initial set of states of the function. One-step concrete pre-image S^1 of $S_M \downarrow$ checks whether any new states can be added to $S_M \downarrow$. If $S^1 \setminus S_M \downarrow$ is non-empty then the local abstraction R_f is refined and the loop continues. Otherwise the global abstraction R_G is refined with respect to S_M^f . The local and global refinements are described in the next paragraph. The algorithm terminates when each abstract state can either reach E or can not reach E in one function step.

Algorithm 2 AbsRef(R, f, E)

Input: Abstraction R , function f , error set E **Output:** updated R

1. $V_{abs} := \cup_{r \in R} support(r)$, $R_f := R$
 2. **loop**
 3. $S_M := Pre_M^{f, R_f}(E)$; $S_M^f := S_M \cap I_L^f$
 4. $S^1 := Pre^{f, 1}(S_M \downarrow)$
 5. $s_{new} := S^1 \setminus (S_M \downarrow)$
 6. **if** $s_{new} := \emptyset$ **then** $R_G := R$
 7. **for each** $r \in R$ **do**
 8. **if** $(r \cap S_M^f) \neq \emptyset$ & $(r \setminus S_M^f) \neq \emptyset$
 9. $R_G := R_G \cup \{r_1, r_2\} \setminus \{r\}$, where $r_1 := (r \cap S_M^f)$ and $r_2 := (r \setminus S_M^f)$
 8. **return** R_G
 7. **else**
 8. split including a variable v from $\{v \in (V^f \setminus V_{abs}) \mid v \in support(s_{new})\}$
 10. Abstraction R_f is refined for all valuations of v
 11. **end if**
-

Automatic Refinement : For refinement of the local abstraction R_f , the algorithm finds a variable $v \in V^f$ which is not in the set V_{abs} and is in the support set of $S_M^1 \setminus S_M \downarrow$. The variable is added to the significant set V_{abs} and a new abstraction R^f is obtained with respect to different valuations of v . The refinement of global abstraction R_G happens after the local abstraction reaches a fix-point and no new states can be added in the S_M set. For each abstract state $r \in R_G$ have a non-empty intersection with both S_M^f and $\neg S_M^f$, then it is split into two states r_1 and r_2 .

Algorithm 3 BuildInterface(R, F, Lib)

Input: Abstraction R , a set of functions F , a library $Lib = (F_G, V_G, E, I)$ **Output:** Interface Graph $IG = (N, T, T_e, In, Er)$

1. $Q, N, T, T_e, In, Er = \emptyset$
 2. $append(Q, I)$; $append(N, I \cup E)$; $append(In, I)$; $append(Er, E)$
 3. **while** Q is non-empty **do**
 4. $curr := removeFirst(Q)$
 5. **for each** $f \in F$ **do**
 6. $next := Post_m^{f, R}(curr)$
 7. **if** (**not** member(N , $next$)) **then** $append(Q, next)$; $append(N, next)$ **endif**
 8. **if** ($next \subseteq E$) **then** $T_e := T_e \cup (curr, f, Er)$ **else** $T := T \cup (curr, f, next)$ **endif**
 9. **end for**
 10. **end while**
-

Building Interface : Algorithm 3 computes the interface graph from the abstraction R . For the algorithm, a list Q is maintained. the procedure $append(Q, X)$ adds each element $x \in X$ at the end of Q . The procedure $member(Q, x)$ check if x is a member of Q . The procedure $removeFirst(Q)$ removes the first element from Q and returns the element. The algorithm computes the next symbolic state for each element in Q by applying $Post_m^{f,R}$ operator. There is an error-edge from the current state $curr$ to the error state Er when the next state of $curr$ is a part of error set E . Otherwise appends the next state Q and a new good edge $(curr, f, next)$ is added. The algorithm terminates when the list Q is empty.

Example 2. To illustrate the algorithms defined before, let us revisit the Integer Stack example (Figure 1). We assume that the guarded-update rules (Figure 1(b)) are converted into a library model with the set of functions $\{pop, push\}$. Let us denote the state-space as S . Figure 2 illustrates the run of the explore algorithm (Algorithm 1). The initial abstract states r_0, r_1 and r_2 partitions the state-space S into three regions (Figure 2(a)), where $r_0 = S|_{err=1}$ corresponds to error states, $r_1 = S|_{err=0, top=0}$ corresponds to the initial states without error states, $r_2 = S|_{err=0, top>0}$ corresponds to the non-initial non-error states. *AbsRef* (Algorithm 2) is invoked for pop function, the significant variables are

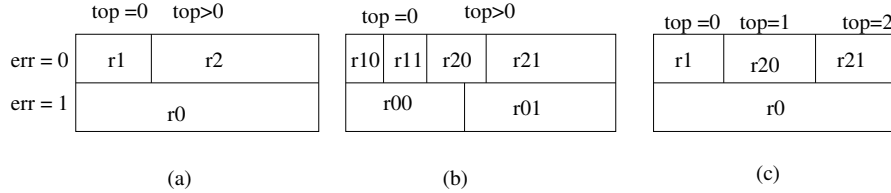


Fig. 2. Run of the algorithm Explore on IntStack Example. (a) The initial abstraction (b) The local abstraction inside function (c) The final global abstraction.

$V_{abs} := \{err, top\}$. In the first iteration, the must predecessor S_M of error state r_0 fail to add any new states. However, one step concrete predecessor of set S_M returns a set S^1 corresponding to $S|_{pop.s=0, top=0, err=0}$, where $pop.s$ is the local variable s at function pop . The support set of $S^1 \setminus S_M$ contains a new variable $pop.s$ which is in V^f , but not in V_{abs} . The local refinement of R_f adds different valuations of local variable $pop.s$ (Figure 2(b)). The second digit of each abstract states denotes the value of $pop.s$ in the abstract state. In the next iteration the must predecessor S_M becomes $\{r10, r00, r01\}$ and no new concrete states can be added by one step predecessor of set S_M . Hence the local abstraction R_f can not be further refined. The local refinement at Figure 2(b) can not be returned as the locally added variable $pop.s$ can not reach outside the scope of function pop . The global set which leads the error set can be given by S_M^f which is a subset of S_M corresponding to local initial state I_L^f of the pop function i.e. $S|_{pop.s=0}$.

Hence the final global abstraction R_G for pop function is obtained from the initial global abstraction R of the function and will be refined with respect to set S_M^f and its complement set. The algorithm returns with an unchanged global abstraction.

Similarly for the push function the local variable `push.s` is included in the local abstraction. Even if no new global variable is added in the refinement, there is a new refinement of the global abstract set r_2 with respect to the set of states (where `top` is 2 and `err` is 0) which reaches error states in one push call. The final global abstraction is shown in Figure 2(c). The build interface algorithm (Algorithm 3) starts with the initial state r_1 and adds the edges in the graph (Figure 1(c)) until every node is explored with respect to all functions.

The interface generated by Explore algorithm is safe and permissive by construction. The safety is ensured by *AbsRef* Algorithm and permissiveness is ensured by *BuildInterface* algorithm. The final abstraction R after calling *AbsRef* algorithms for each function $f \in F$ distinguishes error reaching regions from the non-reaching ones. In *BuildInterface* algorithm each function f is applied in each of the states in the graph obtained by the abstraction R and hence all behaviors are captured in the interface graph.

Theorem 1. *Explore (Algorithm 1) returns a safe and permissive interface.*

4.2 Implementation Optimizations

Approximate Abstract Function Summary and Predecessors: For practical purposes, we do not compute the abstract predecessor operators on the monolithic transition relations. Like [7], Equation 4 holds for approximate operators. The transition for a function $f \in F_G$ is represented as a number (say k) of guarded-update rules. For an abstraction $R \subseteq 2^{2^{S_f}}$, the must and may abstraction of rule $i \in \{1, \dots, k\}$ can be given as follows:

$$\begin{aligned} i.trans_{m+}^{f,R} &:= \{(r_1, r_2) \in (R \times R) \mid r_1 \in i.guard \uparrow_R^m, r_2 \in i.update(r_1 \downarrow) \uparrow_R^m\} \\ i.trans_{M-}^{f,R} &:= \{(r_1, r_2) \in (R \times R) \mid r_1 \in i.guard \uparrow_R^M, r_2 \in i.update(r_1 \downarrow) \uparrow_R^m\} \end{aligned}$$

For all $j \in \{m+, M-\}$, $X \subseteq 2^R$, the approximate transition relation, one step predecessor operator and multi-step predecessor operator can be given respectively as:

$$\begin{aligned} Trans_j^{f,R} &:= \bigcup_{i=1 \dots k} i.trans_j^{f,R} \\ Pre_j^{f,R,1}(X) &:= \{r \in R \mid Trans_j^{f,R}(r) \cap X \neq \emptyset\} \\ Pre_j^{f,R}(X) &:= \{r \in R \mid r \cap (\mu Y.(X \cup Pre_j^{f,R,1}(Y))) \neq \emptyset\} \end{aligned}$$

. For disjunctive transition relation, the approximate may predecessor operator will be precise; however, the approximate must predecessor will be under-approximation of the precise one.

Theorem 2. For each $f \in F$, $R \subseteq 2^{2^{S_f}}$, and $X \subseteq 2^R$, we have

$$Pre_{M-}^{f,R}(X) \downarrow \subseteq Pre^{f,*}(X \downarrow) \subseteq Pre_{m+}^{f,R}(X) \downarrow.$$

Incremental Building of Interface: Algorithm 1 can be used for incremental addition of function sets; as we may not need to create the interface for all the functions at first. The algorithm returns the refined interface for the included functions only. The created interface can be used if we want to add more functions from the library.

Rule Partition for Function One more optimization will be partitioning the rule set of each function with respect to the abstraction to create *less splitting*. Computation of each individual rule for must abstraction can create huge under-approximation; hence may need more splitting.

Example 3. In presence of If-Then-Else or Switch constructs in the source code, we may encounter the following rules after the translation.

$$\begin{aligned} r_1 : hd = true &==> indata' = 0; hd' = false \\ r_2 : hd = false &==> indata' = 0; hd' = hd \end{aligned}$$

The abstract set R is defined with respect to different valuations of $indata$ variable. If we consider each rule separately and apply the must abstraction, we miss the fact that the final value of variable $indata$ will be 0 and does not depend on the initial value of hd . The must predecessor of $S|_{indata=0}$ will be \emptyset for both rules since the must abstraction of guards will be empty-set. However, if we combine two rules by taking union of sets, then the must predecessor of $S|_{indata=0}$ will be S for the combined rule and there will not be any further splitting.

The heuristic of rule set partition is obtained from the abstraction itself. If a function f has k rules, then i -th and j -th rules can be grouped together for an abstraction R if the condition $i.guard \uparrow_R^m = j.guard \uparrow_R^m$ holds.

5 Results

In this section we will provide results of some case studies and compare with the related works.

Data Stream Case Study There is a data stream with a header of length 2^h and data of length 2^d where $h \leq d$. The program uses d bits to represent the pointer and 1 bit for the "error". The boolean variable *isHeader* is 1 when in header and is 0 otherwise. There are four functions in the program. The function *FirstHeader* and *FirstData* takes the pointer to the first header and data location respectively. The function *Next* moves the pointer within the header or data in a cyclic way. The function *Write* results in an error when pointer points to header section. Our algorithm produces the interface shown in Figure 3(a). The state 1 represents that the pointer in the data part and the state 2 represents that the pointer in the header part.

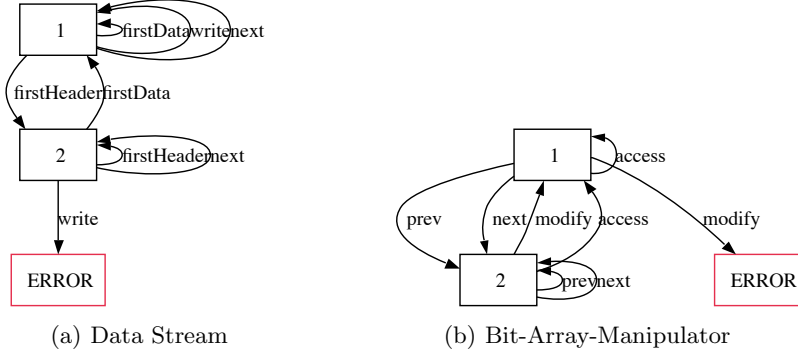


Fig. 3. Interfaces

Bit Array Manipulator The Bit Array Manipulator has four functions : *prev* , *next*, *access* and *modify*. Two global variables *ptr* of length 2^k specify the current location of the pointer. The global Boolean variable *valid* denotes whether the pointer is valid. Another Boolean variable *err* specify the library error states. The functions *next* and *prev* respectively increments and decrements the current pointer and set the valid flag to true. The functions *access* resets the valid flag. The function *modify* return sets *err* to true when the valid is false, otherwise sets valid to false. Our algorithm produces the interface shown in Figure 3(b). The state 1 represents that the valid bit is false and the state 2 represents that the valid bit is true.

Case Study	Params	Time (ms)	Regions	Direct	Learning	CEGAR
Data Stream	$h = 2, d = 12$	3	2	1028	2	257
	$h = 4, d = 12$	4	2	4112	2	257
	$h = 13, d = 13$	18	2	16384	2	2
Bit Array Manipulator	$k = 8$	2	2	68	2	2
	$k = 9$	4	2	130	2	2
	$k = 16$	8	2	16386	Timeout	2

Fig. 4. Results

Comparison Figure 4 shows a comparison of our algorithm with the related work on these two examples. The first two columns show the name and different parameter values of the case-studies. The next column describes the running time (in milli seconds) of explore algorithm from the parsed guarded-update rules. The next column represent the number of non-error regions in the interface graph. The non-error regions from other three related work are given in the last three columns and the data is obtained from Beyer et. al.'s. work [2]. The

results for Direct algorithm show that direct algorithm runs fastest, but the size of interface graph is exponential in d . We obtain that the CEGAR algorithm provides minimal graph only when $h = d$ in the Data Stream example. The size of the graph in the CEGAR algorithm depends on the proper representation of variables with Boolean variables. The CEGAR approach refine by adding a new boolean variable; which has a risk of splitting many abstract states unnecessarily. In contrast, our algorithm keeps global abstraction separate from local abstraction inside the function and refines the global abstraction lazily with respect to the final reachable set (S_M^f). Learning algorithm provides the minimal graph, but slowest of all three approaches. Our algorithm provides the same number of non-error regions as the learning algorithm. However, we can not compare time due to different platforms.

6 Application of Interfaces

In this section, we show how a safe and permissive interface can be useful in the verification and testing of the software programs. The following section briefly describe the modifications needed for the interface to be compatible with these settings.

6.1 Software Verification with Interfaces

Let us assume that we have computed an interface graph for a set of functions. Given a client program consisting of those functions one can immediately check the client with respect to the interface graph. The idea would be simulating the actions of the client program into the interface graph and check whether the library error state (State "ERROR") is reached. For example, a client with a single line *modify(b)* on the BitArrayManipulator b can be simulated in the interface graph (Figure 3(b)). We can see that the error state ERROR is reached from the initial state (State 1). There could be an infinite number of possible clients corresponding to those functions and each of them can be model-checked after the interface is computed.

6.2 Offline Test Case Generation

In the model-based testing paradigm, an implementation under test (IUT) is checked with respect to a given model program (a specification of the IUT). Our algorithm can build an interface graph from the definitions of the functions given in the model program. We can create a C source regression test-suite from the interface generated from the libraries. However, we need to extend the function calls with the argument values to create a test-bench for the IUT. For example, Figure1(a) can be generated from the model program in Figure1(c). If we are given a linked-list implementation of a finite-size integer stack, we can create an offline test-suite from the interface graph. The testing of the implementation with respect to the test-suite checks whether the interface goes to the error state

if and only if the implementation goes to the error state. If there is a discrepancy between the behavior of the interface graph and the code, we understand the implementation source needs further checking.

7 Conclusions

In this section we conclude with the summary of the work and possible future directions. We have provided a new algorithm for interface synthesis with a local-global abstraction refinement framework. This framework is can dramatically reduce the state-space of the interface generation by hiding local variables inside each function. The abstract summarization of the functions provides scalability. The modular analysis is used to handle each function separately. In our generalized setting any C-style set of functions can be handled.

The results show that our algorithm provides a safe, permissive and sufficiently minimal (i.e. comparable to the learning algorithms) interface from the set of functions. We have provided the approximate abstract predecessor operators to handle the state-space inside the function. The interface synthesis can be incremental : hence one can add new functions to the interface and it may lead to refinements corresponding to the function.

The interface could be used to immediately verify clients and as offline test-suite for a new untested implementation. However, the translation engine is very basic and some parts are done manually. In future we like to work more on covering more aspects (e.g. pointers, recursive data types) of the C source code such that we can have bigger case studies. We like to see how we can use the shape analysis algorithms to translate complex data types. We also like to include CIL inside the tool TICC s.t. it can parse C functions and represent the rules directly in MDD format. We like to implement the back-end using a combination of MDD and SMT solvers such that the space-space problems can be handled better.

References

1. R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, 2005.
2. D. Beyer, T. A. Henzinger, and V. Singh. Three Algorithms for Interface Synthesis: A Comparative Study. Technical report, 2006.
3. D. Beyer, T. A. Henzinger, and V. Singh. Algorithms for interface synthesis. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, July 3-7)*, LNCS 4590, pages 4–19. Springer-Verlag, Berlin, 2007.
4. L. de Alfaro, B. Ader, M. Faella, A. Legay, V. Raman, P. Roy, and L. Dias Da Silva. TICC: Tool for interface compatibility checking, 2006. <http://dvlab.cse.ucsc.edu/dvlab/Ticc>.
5. L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *FRODOS: Frontiers of Combining Systems, Proc. of the 5th Intl. Workshop*, volume 3717 of *Lect. Notes in Comp. Sci.*, pages 81–105. Springer-Verlag, 2005.

6. L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proc. 19th IEEE Symp. Logic in Comp. Sci.*, pages 170–179, 2004.
7. L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2007.
8. E. Emerson and C. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *Proc. 32nd IEEE Symp. Found. of Comp. Sci.*, pages 368–377. IEEE Computer Society Press, 1991.
9. T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 31–40, New York, NY, USA, 2005. ACM.
10. T. Kam and R. Brayton. Multi-valued decision diagram, 1990. UC Berkeley Electronics Research Laboratory, Memorandum No. UCB/ERL M90/125.
11. G. Necula, S. McPeak, W. Weimer, R. To, and A. Bhargava. CIL: Infrastructure for C program analysis and transformation.

Appendix

A C function to compute n-th Fibonacci number is translated into a set of guard-update rules. To handle the activation stack and store the context of the caller, there is an explicit implementation of integer stack. The variable *nextpc* denotes the next value of the location variable after return from one of the the stack operations. The variable *v* contains value of input parameter of push and is assigned before a call to push . *v* is the output parameter of pop and obtained after returns from pop.

```

module Fibonacci:
  var i,s,top : [0..MAX]
  var v:[0..15]
  var a0, a1, ... : [0..15]
  var nextpc: [0..31]
output push: {
  s=15 & top < MAX ==> top'=top+1 & i'=top & s'=16;
  s=16 & i=0 ==> s'=nextpc & a0'=v;
  .....
}
output pop :{
  s=17 ==> i'=top & t'=18;
  s=18 & i=0 ==> s'=19 & v' = a0;
  .....
  s=19 & i>0 ==> top'=i-1 & s' = nextpc
}
...
endmodule

```

The rule set *fib* defines the transitions inside the Fibonacci function. The variable *res* stores the result when the call returns and *tmp1* and *tmp2* are two temporary

variables. A recursive call to itself is translated into saving the return address, the current value of n , initializing n for the called function and a subsequent jump to the initial location of the function.

```
var n : [0...20]
var res, tmp1, tmp2 : [0..31]
output fib: {
  s=0 & n<3 ==> res'=1 & s'=11;
  s=0 & n>=3 ==> s'=2;
  s=2 ==> nextpc' = 3 & s'=15 & v'=5;
  s=3 ==> nextpc' = 4 & s'=15 & v' =n;
  s=4 ==> n' = n -1 & s'=0;
  s=5 ==> t'=6 & tmp1' = res;
  s=6 ==> nextpc' = 7 & s'=15 & v'=9;
  s=7 ==> nextpc' = 8 & s'=15 & v'=n;
  s=8 ==> n'=n-2 & s'=0;
  s=9 ==> s'=10 & tmp2'= res;
  s=10 ==> s'=11 & res' = tmp1+tmp2;
  s=11 ==> nextpc' = 12 & s'=17;
  s=12 ==> n' = v & s'=13;
  s=13 ==> nextpc' = 14 & s'=15;
  s=14 ==> s' = v;
}
```